Chapter 3

Software Development Life Cycle Models

In this chapter, we describe the over-arching processes in software engineering: *software development life cycles*. Some example SDLCs were presented in Chapter 2. In this section, we explore these and other lifecycles in more detail.

3.1 Software Systems

Software systems are ubiquitous[1] and is found in everything from home appliances to space shuttles. It is integral to cars, trains, planes, ships, communications systems, mobile phones, command and control centres (such as police, fire and ambulance), to government and to business and much of the fabric of our society depends on software.

Software systems are also becoming more complicated[2]. The types of systems that we are talking about are composed of many interconnected modules, whose behaviours are typically both variable and dependent on the behaviour of the other parts. The real difficulty in understanding such systems is that the behaviour of the whole is highly dependent on the individual parts, and each part is highly dependent on the other parts.

3.1.1 Traditional vs Modern Software Systems

Up until recently, software was mostly designed to run as standalone applications that run on desktops or simple client-server applications. These applications were developed by software engineering teams, packaged and deployed to be installed at the customer sites and were managed by the system administrators who are also IT professionals, in the customer organization. However, this paradigm has changed recently with most of the applications being hosted in the *cloud*, and accessed by a client applications, which take the form of mobile or browser-based applications. Software that is embedded in devices are also still common, but they also operate differently from traditional applications. Following are some common types of modern software applications [KMR16].

- **Hosted Applications:** These applications run on servers, that are managed by the developers themselves, and users access the application as software as a service (SAS) model. The software may run on a dedicated set of servers owned by the service provider (e.g. Google, Facebook), or they may be hosted in third party cloud computing services such as Amazon. One of the major advantages of this paradigm is the level of control the application developers have; application developers can change the server software as they wish because this does not require sending updates to customers.
- **Mobile Applications:** These are programs that are installed on mobile devices. Such application generally access a remote server for data and business logic, while the application running on the user device provides the basic user interface. Users download and install such applications through an application store, and application

[1] *Ubiquitous*: appearing or found everywhere.

[2] Many authors assert something similar by saying that software systems are becoming more complex. The phrase *complex system* however, has a technical meaning, which is why we avoid the term "*complex*" here.

developers upload new versions of the software to a application store. Often the developers have to comply with a some type of process to upload new versions of the software to the application store.

- **Embedded Applications:** These are applications that run on electronics devices, such as wearables, automobiles, and medical devices. Traditionally this type of software was installed once and did not change during the lifetime of the device. However, embedded software available modern devices can be updated through over-the-air (OTA) update capabilities, and therefore, today this type of software is also deployed and managed different from the traditional embedded software.

3.1.2 Engineering Software Systems

Recall that software is *intangible* and the algorithms and data structures making up the key elements of software systems are creations of logic. All that we can see of software is some representation of it; for example, in the form of program listings, or in the form of more abstract representations such as UML. Our understanding of the behaviour of a software system depends on our understanding of the computer and the manner in which it executes the code.

To date, the only real way we have of engineering software is through *software engineering process*. There are a number of competing philosophies about how software should be developed but for now we give only two of the most salient.

Formal software engineering is the philosophy that we need to exercise control over all aspects of the project. Formal processes are sometimes called the *prescriptive processes* because they prescribe what must be done in order to achieve project outcomes.

Agile software engineering is the philosophy that we need to primarily exercise control over the outcomes of a project and guarantee that what is developed satisfies the outcomes. Agile processes are sometimes called the *reactive processes* because they react to the project circumstances.

When thinking about processes we often talk about the *process models*.

Definition 3.1. A *process model* is a template for a process that shows the generic activities, inputs and outputs.

A process model describes the ideal process that one should follow to achieve a goal. In software engineering, the ideal process is rarely followed; developing software is quite difficult, so to apply a process with everything going exactly to plan is highly unusual.

3.2 Overview

A *software life cycle model* (SDLC) describes the life cycle of a software project, from conception through to maintenance. It provides a structured blueprint for how a software project should progress. There is no software life cycle model that fits all projects, but experience and empirical evidence helps us decide which life cycle models are good for certain types of projects.

A software project usually involves the following *activities* or *phases*:

Requirements engineering: eliciting the requirements for the system, analysing and defining these requirements using models, and validating the requirements.

System/architectural design: defining the sub-systems (or components) within the system, and the relationships between them.

Detailed design: defining the behaviour of the components to fulfil the functional requirements.

Implementation: programming the detailed design in some language.

Integration: integrating the programmed parts with respect to the architecture.

Testing: running executable tests on the programs in an attempt to find faults. Testing can take several forms, including:

unit testing: tests the behaviour of individual components against their design;

integration testing: tests the behaviour of the sub-systems of components against the design;

system testing: tests the behaviour of the system as a whole against the requirements; and

acceptance testing: tests that the system as a whole against the expectations of the customer or user.

Delivery and release: packaging the software in an accessible manner and with useful documentation.

Maintenance: modifying the software to fix faults or to provide new requirements — sometimes a new project life cycle in its own right.

Each of these phases produces *artifacts*, many of which are used as inputs to other phases. For example, the output from the requirements engineering phase is a *software requirements specification* (SRS), which is used to inform the design, to generate test cases, to produce user documentation, and to maintain the system.

There is no strict order on these phases; different SDLC models offer different orderings and rationales for these orderings. For example, while testing is performed on the implemented software, some lifecycle models specify that the test phase begins before implementation, as some test cases can be defined as soon as the software requirements specification is complete.

Typically, when designing a process to meet some goal we would start with the *lifecycle* model and then add in any activities for special requirements. For example, if we require high reliability then we may begin with an iterative process, but add prototyping, random testing, and reliability measurement to the generic template, such as the modified iterative process model in Figure 3.1.

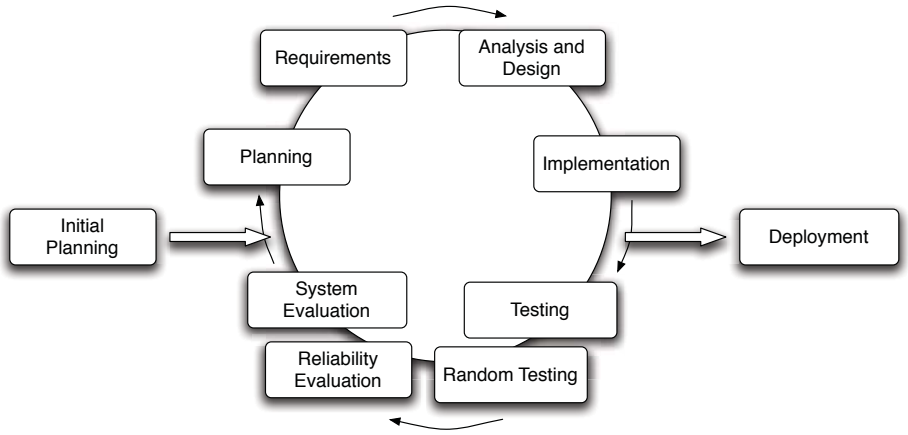

Figure 3.1: Changing the iterative process model by adding in reliability testing and evaluation.

The next step is to think about each of the activities in turn and ask the following questions:

- **What steps do we need to take to produce the outputs from the inputs?** We have already seen an example in requirements analysis where we can break the requirements activity into elicitation, analysis, specification and validation activities. Figure 3.2 shows some of the possible sub-processes and techniques that can be applied to achieve the goals of each major process phase. Generally, we can identify two types of processes: *within phase* and *across phase* processes. Much of the study of software engineering is in the processes, tools and techniques required to develop specific kinds of system. In this introductory subject we study the *within phase* processes and explore their impact on projects.

- **What techniques and tools can be applied?** Again in the requirements example we will see many special techniques for requirements elicitation; for example, interviewing, brainstorming, and focus groups to name a few.

Each phase in a software engineering process is designed to take inputs and produce outputs that will be used in subsequent phases. *Tools and techniques* enter the picture at this point because to produce the outputs for most software engineering phases we often require deep analysis (using the *techniques*) and the production of complicated outputs (using the *tools*). Tools are also used to help automate the more mundane steps in the production of an output and to manage the flow of artifacts between different process activities.

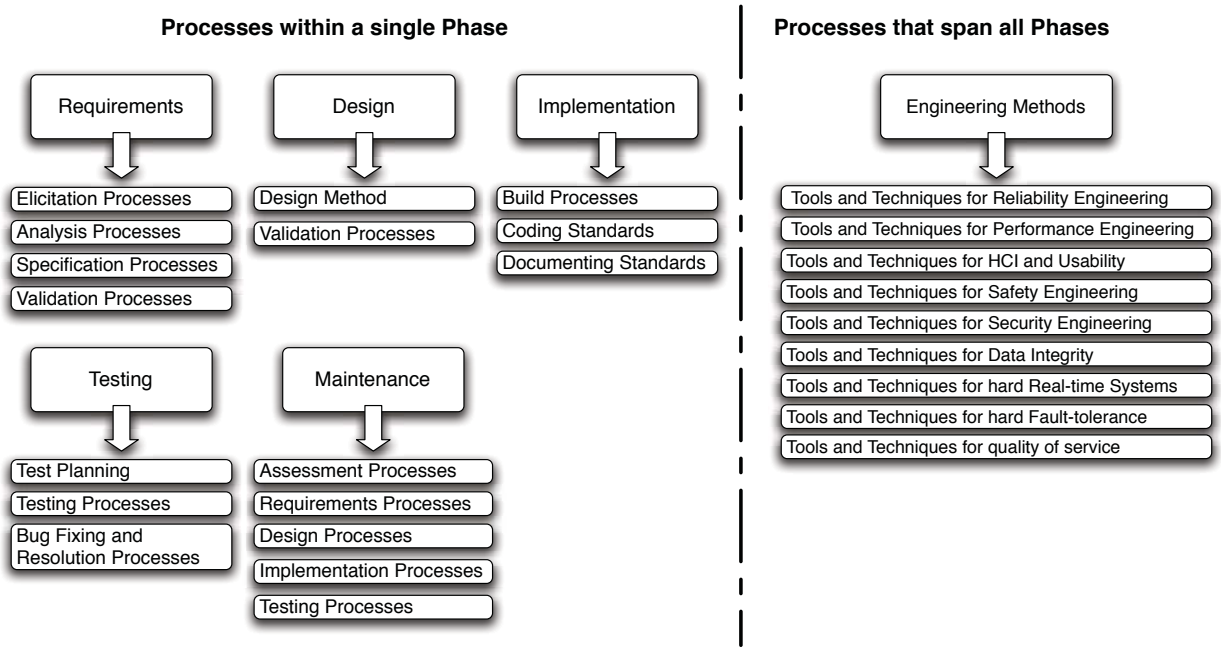

Figure 3.2: Types of sub-processes and techniques to achieve goals of a major process.

Example 3.1. *Sub-processes within the development lifecycle*

1. The requirements phase takes, as input, any sources of requirements including project briefs, stakeholder lists, concept documents, and documented knowledge of prior similar systems, and produces a *software requirements specification* (SRS).

   SRSs are usually delivered in the form of a *requirements package* — a combination of English language requirements, models, mockups, and prototypes — and are complicated artifacts because of the high number of interdependencies and the need for consistency across the entire package.

2. The design phase takes as input a set of requirements — it does not have to be complete — and produces a design. It may be an architectural design or a detailed design, and must be passed on to an implementation team who *build* the design.

   Designs are complicated because we typically build complicated systems. They are also produced at many different levels of abstraction; for example: (1) architectural designs that specify the main parts of the system, how they are interconnected and how they collaborate to achieve the main goals of the system; or (2) detailed designs that specify what functions are present in a component and the function preconditions and postconditions.

3.3 Formal Software Development Lifecycle Models

In this section, we introduce some of the most well-known formal SDLCs, and explore their strengths and weaknesses.

The Waterfall Model

The waterfall model is one of the oldest and most venerable process models whose history goes back to the paper by Royce [Roy70]. Royce, however, did not actually advocate that the lifecycle phases in his paper should be a process for software development. In fact, he proposed the waterfall method as a flawed model of a SDLC. His analysis came to the conclusion that whatever process was actually used it should contain at least a *requirements analysis* phase, a *design* phase, a *coding* phase, a *testing* phase and a *maintenance* phase. For Royce at least, these stages could be combined in different ways to create more effective processes.

Royce's collection of phases later became the basis of a process and, together with its variations, has been much used. The basic waterfall assumes that each major phase can be completed and the outputs reviewed and worked up to an acceptable standard *before* moving on to the next phase.

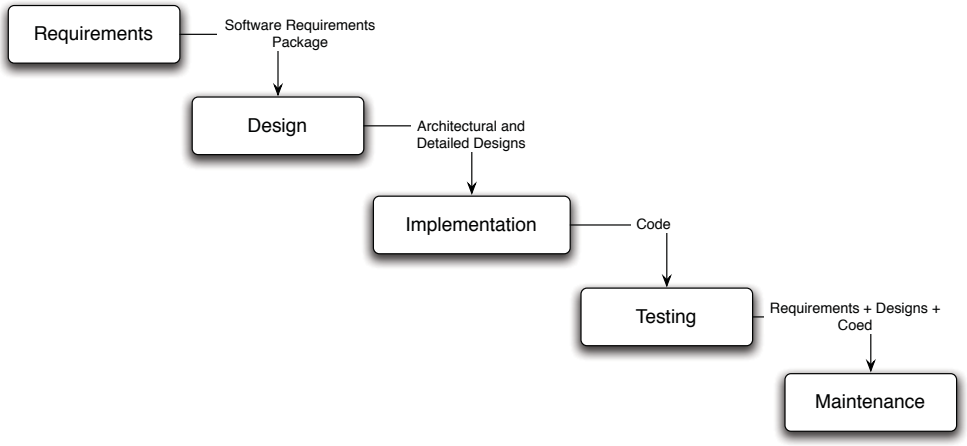

Figure 3.3: The Waterfall Model

In the pure waterfall model, shown in Figure 3.3, we complete the requirements phase to produce a *software requirements package* that becomes the input to the design phase. The requirements phase may itself be broken down into elicitation, analysis, specification and validation sub-phases as in Figure 3.4, but in a pure waterfall all of these are completed before moving onto the design phase.

A very important aspect of the waterfall model is that each major phase within the model is concerned with producing an artifact. These are:

- The *requirements package* produced in the requirements phase;
- The *software design* produced in the design phase;
- The *code base* produced in the implementation phase; and
- The *test plan, test cases* and *test reports* produced in the testing phase.

These artifacts are important to the waterfall process because of two main reasons:

1. They allow developers to *measure progress*; and
2. They allow developers to *evaluate quality*.

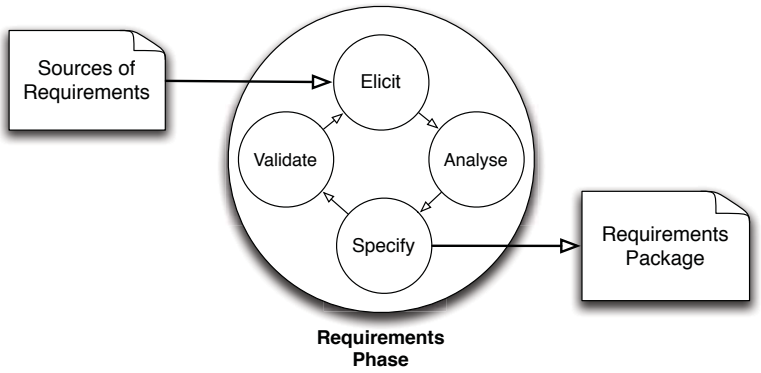

Figure 3.4: A decomposition of the requirements phase into sub-phases.

In practice, however, it is not always possible to complete a phase before moving onto the next. A classic example is architectural design which is often started before requirements analysis is complete. Supporters of the waterfall model claim that is provides good support for projects that are well understood and have clear goals and that it provides a good model for estimating project costs and tracking progress.

The modifications to the waterfall typically allow phases to be revisited one or more times based on feedback from later phases. For example, in a *modified waterfall*, shown in Figure 3.5, an unforeseen technical constraint at the design stage may force additional investigation of the requirements. In this case the requirements phase would be revisited, completed and reviewed again before passing the modified requirements package to the design phase again. Also, prototyping at the requirements and design phases may be added in order to improve the developers confidence in the outcomes of these phases.

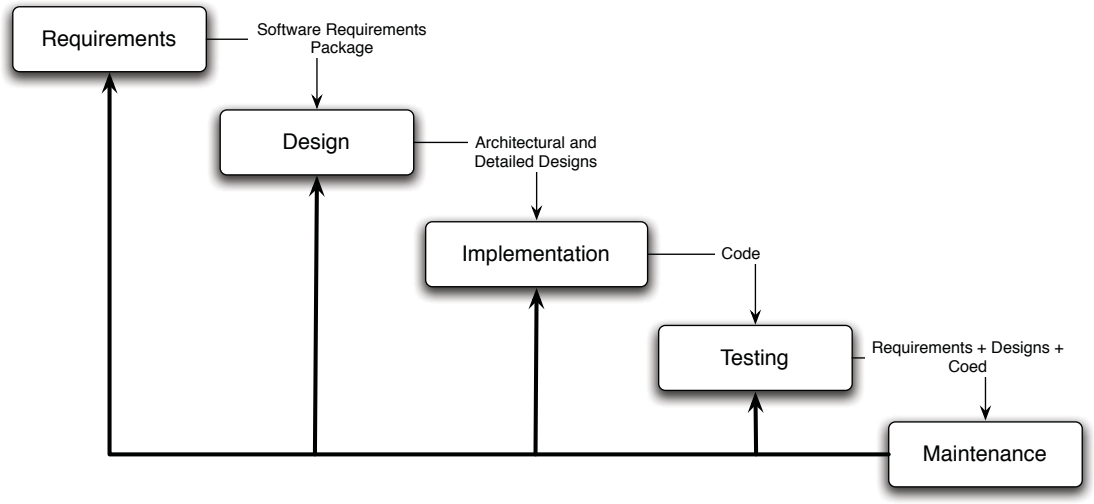

Figure 3.5: A modified waterfall process. Any of the phases may be redone but when a phase is revisited it is completed in its entirety before going back down the waterfall. Thus, if we were to revisit the requirements phase after testing we would need to redo the design phase and the implementation phase.

Detractors of the waterfall model say that it does not take into account the technological and domain related risks that are often found in software projects. Also, waterfall models do not allow for the iterations that are often required as understanding of the problem domain evolves during the course of the project. Further, for large and complex

projects there is not enough feedback provided to the client and it may be months (or years) between commissioning a system and seeing some part of it working.

In general, when using a waterfall model, it is better that the outputs of any phase remain relatively stable throughout the project once they are completed. Revisiting phases too often in a waterfall slows the project down and in situations where requirements are uncertain or where technology may change a different process is better suited.

The V Model

The *V-model* is a development model that originated from the need to develop better testing processes. The *V* model shows exactly which testing activities correspond to each of the analysis (concept and requirements) and decomposition (architectural and detailed design) activities. Thus we perform unit testing by developing test cases from the detailed design and verifying the results of a test against the detailed design. Similarly we perform system testing by developing test cases from requirements and verifying the results of tests against requirements (see Figure 3.6).

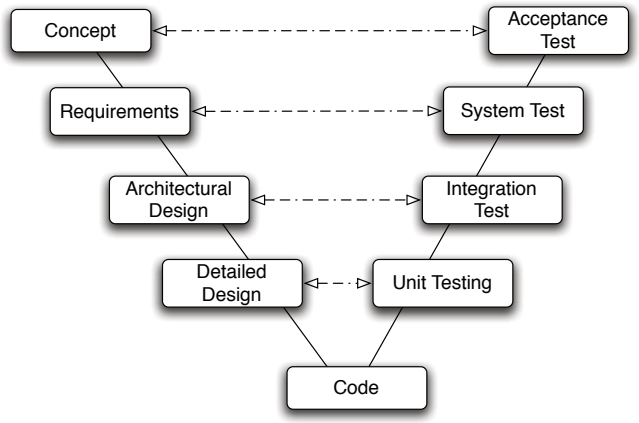

Figure 3.6: The V model of software development, which matches up development stages with assurance activities.

To see how the *V*-model might work in practice suppose that we are dealing with a language like Java where the smallest unit of development is a *class* and so in unit testing we will be testing classes. To test a Java class without have built system, we need a *driver* — a program that will supply values to the methods in the class being tested and to capture the outputs of those methods.

In practice the test plan and the test cases can be written in the phase corresponding to the level of testing required. For example, the test-plan and test-cases for systems testing can be written in the requirements phase.

Incremental and Iterative Development Models

Incremental and iterative models aim to deal with uncertainty and changing project environments.

Incremental models divide the development into a fixed number of increments each involving a *planning*, *requirements*, *design*, *implementation* and *testing* phases. Each increment may follow a mini-waterfall or may even follow some other process. Figure 3.7 shows the desired process to be followed when using incremental development.

The key requirement for each increment is that it develops a complete and usable subset of the system functionality that can be deployed and can be evaluated. Changes required by the evaluation forms part of the planning and requirements for the next iteration.

While incremental delivery offers more flexibility than a waterfall and reduces many of the risks of a changing environment, the process needs to well managed.

1. The first planning and requirements phase needs to provide a detailed enough set of requirements so that at least a high level design for the system can be done and the functionality divided into increments.

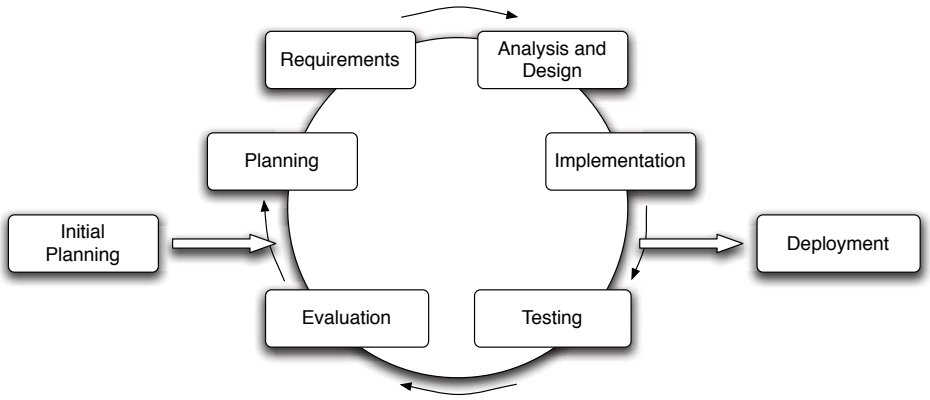

Figure 3.7: Incremental development

2. A software architecture often (but not always) needs to be produced early in the process and used to integrate increments.

3. Each evaluation needs to be turned into a set of requirements for the next increment. If requirements are changing then we must also keep an accurate record of which requirements pertain to which increment.

4. Typically good configuration management procedures need to be put in place before commencing incremental development projects.

Incremental models manage the risk of changing and uncertain environments by aiming to release often and early in order to receive client and user feedback on the evolving product. In the case of unfamiliar or changing technology developers can correct earlier misunderstandings in their design and their code as they learn more about how to get the most from the technology. In the case of uncertain or changing requirements developers can learn more about the problem domain and, as they learn more, correct or improve on their earlier work.

An iterative process is one in which the development is broken up into a number of iterations. Each iteration has the purpose of:

1. Refining and improving the requirements, design and implementation of the system based on feedback from users and testing;

2. Adding new functionality to the evolving system.

Iterative processes and incremental processes are similar and are often confused, but they require different practices and techniques.

- Incremental development specifies that the process develops different parts of the system at different times or at different rates and then integrates them as they are completed.

- Iterative development is a reworking development strategy in which time is set aside to revise and improve parts of the system.

Iterative processes manage the risk of changing and uncertain environments by also aiming to gain client and user feedback often and early. In the case of iterative models developers can refine the requirements, design and system at each iteration.

The Spiral Model

The *spiral* model is a type of iterative model in which each iteration has distinctive sequence of activities that are designed to manage risk. The spiral model is also designed to provide opportunities for getting a better understanding of the problem domain throughout the project. Figure 3.8 shows the desired process in the spiral model.

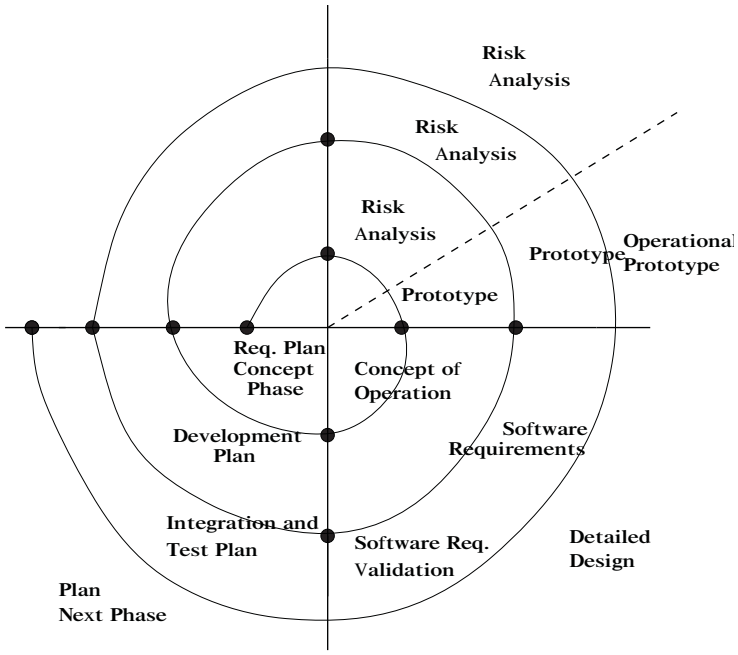

Figure 3.8: The Spiral Model

From Figure 3.8, one can see that the spiral model begins with an initial plan for development and an evaluation of the project risks. The prototyping phases are integral to the spiral model and — as with prototyping in general — are used to understand requirements, design alternatives and to get client feedback early in the development process.

The first prototyping phase is used to evaluate alternatives for the system as a whole and is used to produce the *concept of operations* document that describes how the system should work at a high level.

In the first iteration the sequence of activities is as follows:

1. A prototype of the system is created to explore the system requirements and to explore system alternatives.
2. A concept of operations document is produced from the lessons learned from the prototype that specifies the characteristics of the system and sketches the processes by which it will be built.
3. A set of requirements for the system is produced using the prototype and the concept of operations, and an iteration plan is developed.
4. The design, implementation and testing of the functionality planned for the first spiral now follow.

Note that a *concept of operations* describes the characteristics of the proposed system from the viewpoint of a user of the that system. It is used to communicate the quantitative and qualitative system characteristics to all stakeholders.

Unlike the incremental approach, each iteration typically involves risk analysis, prototyping to determine the feasibility and desirability of various alternatives, and then design, coding and testing.

The spiral model allows for a complete re-evaluation of the project direction after each spiral and this is how risk is managed. Again, there is an issue with managing the spiral model.

3.4 Agile Approaches to Software Engineering

What we have seen so far are formal processes that seek to plan out and to exercise control over the project in order to meet the project goals. Dissatisfaction with the overheads involved in such formal processes has resulted in *agile software engineering methods*. Agile software engineering methods:

- focus on the code rather than the more formal processes;
- are based on an iterative approach to software development; and
- are intended to deliver working software quickly and to evolve the working software quickly to meet changing requirements.

Table 3.1 gives the key principles of the agile approaches to software engineering.

Principle	Description
Customer involvement	The customer should be closely involved throughout the development process. Their role is to provide and prioritise new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People over process	The skills of the development team should be recognised and exploited, rather than simply implementing processes over teams.
Embrace change	Expect the system requirements to change and design the system so that it can accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.

Table 3.1: A table of agile software engineering principles.

The term ***agile*** was formally coined in 2001, in the *Manifesto for Agile Software Development (Agile Manifesto)* (http://agilemanifesto.org/), which emphasises the value of:

- ***Individuals and interactions*** over processes and tools
- ***Working software*** over comprehensive documentation
- ***Customer collaboration*** over contract negotiation
- ***Responding to change*** over following a plan

Agile manifesto is driven by the following principles:

- Customer satisfaction by early and continuous delivery of valuable software
- Welcome changing requirements, even in late development
- Working software is delivered frequently (weeks rather than months)
- Close, daily cooperation between business people and developers
- Projects are built around motivated individuals, who should be trusted

- Face-to-face conversation is the best form of communication (co-location)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Continuous attention to technical excellence and good design
- Simplicitythe art of maximizing the amount of work not doneis essential
- Best architectures, requirements, and designs emerge from self-organizing teams
- Regularly, the team reflects on how to become more effective, and adjusts accordingly

Many agile software development methods exist today, and some of the more popular methods are introduced briefly in the following sections.

Extreme Programming

Extreme Programming (XP) is one of the widely used agile software development methods. Extreme programming is a departure from the formal process models in Section 3.3. It aims to bring people together and focus them on producing a quality piece of software. XP aims to do this by keeping iterations short — two weeks at most — and by running all tests on every build and only accepting a build if all test are passed.

XP uses a set of rules that are designed to respond to change. The principles underpinning the XP techniques are shown in Table 3.2 and the process shown in Figure 3.9.

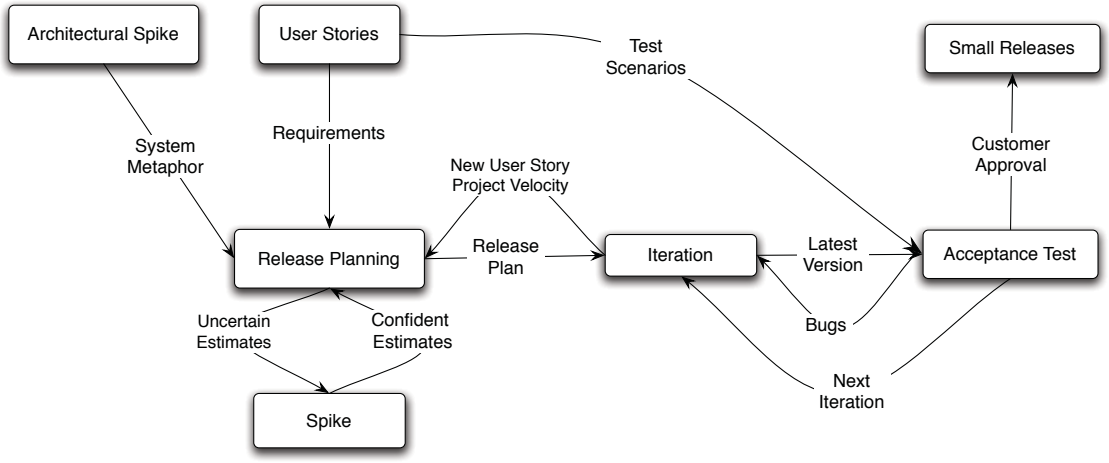

Figure 3.9: The XP flow of activities.

Incremental development is supported by small, frequent system releases. The philosophy is to release early and release often. Clients need to be involved with the project and to be available for evaluating the system when it is released. Ultimately in XP, stakeholders must commit to doing this to keep the cycle times short.

Unlike the formal processes above, it is people, not process, that are the focus of XP. XP uses pair programming, collective ownership and maintains a process that avoids long working hours. Change is supported through regular system releases and client availability. Simplicity is maintained through code re-factoring.

There are some obvious downsides to XP. The most glaring is that the customer is required to provide a representative to the project in a full-time role. Many customers will object to giving up such a valuable resource without having seen prior benefits.

Activity	Description
Incremental planning	Requirements are recorded on *story cards* and the *stories* to be included in a release are determined by the time available and their relative priority. The developers break the set of stories into development tasks.
Small Releases	The minimal useful set of functionality that provides value to the client or stakeholders is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple Design	Enough design is carried out to meet the current requirements and no more.
Test First Development	An automated unit test framework is used to write tests for a new piece of functionality *before* that functionality itself is implemented.
Re-factoring	All developers are expected to re-factor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair Programming	Developers work in pairs, checking each others work and providing the support to always do a good job.
Collective Ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything.
Continuous Integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable Pace	Large amounts of over-time are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site Customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Table 3.2: Principles of XP.

Due to its rejection of formal process, many people incorrectly assume that XP is much easier to follow. However, like formal engineering processes, XP requires a lot of discipline, and is often not followed as it should be. For example, projects struggle to obtain a full-time customer representative, and developers fail to continuously re-factor and integrate their code. Like formal processes, failure to correctly follow the prescribed process makes the process goals more difficult to achieve.

Scrum

Scrum is an iterative, incremental methodology for software development that is becoming more and more used within industry, by far the most popular agile methodology used today. Scrum is a more high-level project lifecycle and management process than XP, however, they share a lot of commonalities, such as short development iterations, called *Sprints* (typically two to four weeks), and being prepared to respond quickly to change.

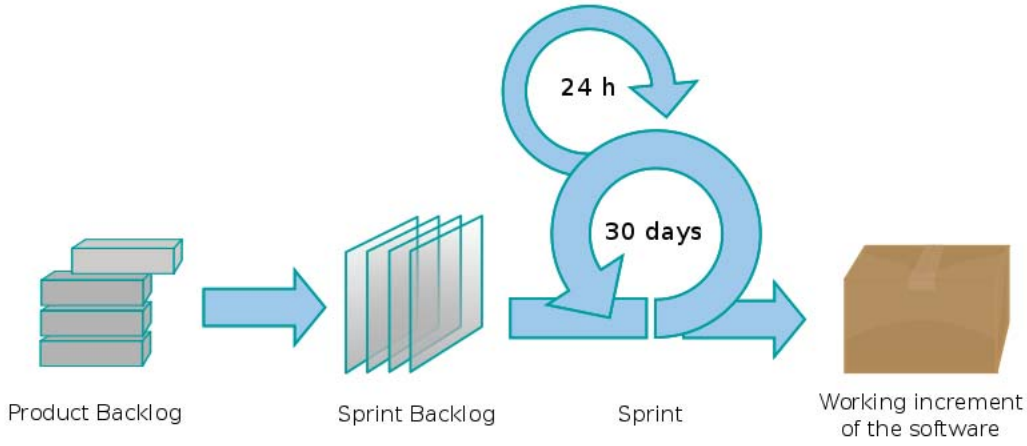

Figure 3.10: A overview of the Scrum development process.

Figure 3.10 shows the development process for a Scrum project. The *Scrum Team* consists of a *Product Owner*, *Scrum Master* and the *Development Team*. The product owner, represents the customer/stake-holders. The product owner writes user requirements in the form of *User Stories*, which are added to a *Product Backlog*, and these requirements are prioritised. At the start of each sprint, during a *Sprint Planning Event*, the team decides which requirements form the product backlog can be implemented in the sprint. This is recorded in the *Sprint Backlog*. The sprint backlog cannot be changed by anyone during the sprint. The product backlog can be changed at any time.

Scrum master facilitates the sprint, and is responsible for product goals and deliverables. He/she is responsible for resolving any impediments the development team might face that could impact the sprint, and also ensures that the Scrum framework is followed. The sprint itself is two to four weeks of development within a single team. During a sprint, the team holds daily meetings, typically held at the start of each day, called the *Daily Scrum*. In this meeting, every member of the team stands up and must answer the following three questions:

1. What did you do yesterday?
2. What will you do today?
3. Are there any impediments in your way?

The purpose of the meeting is for everyone on the team to know what has been done, who has done it, and what remains to be done, while asking team members to make commitments to each other. The meeting has a strict time limit of 15 minutes.

Each entire sprint is timeboxed, typically for two to four weeks, meaning that it must finish at the specified time. That is, if sprints are two weeks long, there is no chance of an extension.

After the sprint is complete, anything from the sprint backlog that is not complete at the end of the sprint goes back to the product backlog. Furthermore, the team hold two meetings:

1. *Sprint review meeting*: the team review the work that was completed and present it to the customer.
2. *Sprint retrospective*: the team reflect on the sprint and discuss what went well during the sprint, and what improvements they can make for future sprints.

Each meeting has a strict time limit, typically three hours.

3.5 The Anatomy of a Process

If we look closely at each phase in one of the formal processes in Section 3.3, we can identify a more general pattern. Each phase typically takes a set of inputs and produces an output (although this is not always true) as in Figure 3.11.

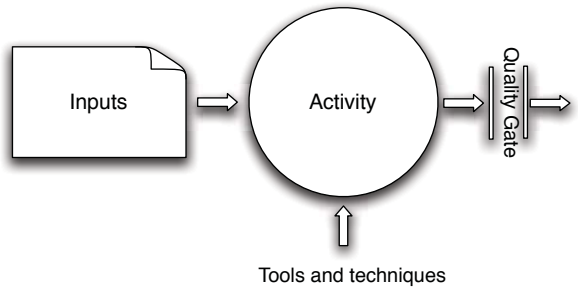

Figure 3.11: A generic phase in a formal process. Note that the activity does not need to produce an output but that in most cases it does. The waterfall process is designed to deliver an output in every major phase so that there is a concrete artifact to measure quality and assess progress.

We have all of the ingredients as before plus one more — a quality gate. The quality gate may be a technical review, or it may be a set of tests to perform on the output. The most important point is that quality is assessed the aim is to ensure that the output of the phase is *fit for purpose*.

As an example, a generic requirements phase may appear as in Figure 3.12. Of course the requirements activity can be decompose further to show the steps in producing the software requirements package from the sources of requirements.

References

[KMR16] Marco Kuhrmann, Jürgen Münch, Ita Richardson, Andreas Rausch, and He Jason Zhang. *Managing Software Process Evolution: Traditional, Agile and Beyond – How to Handle Process Change*. Springer, 2016.

[Roy70] W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, volume 26, pages 1–9, August 1970.

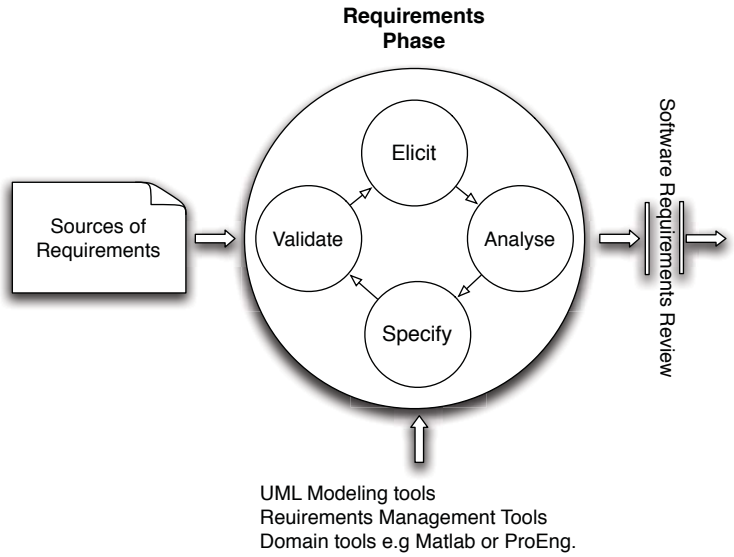

Figure 3.12: An SRS process where some tools and techniques are chosen to help with the process. Note that we have chose a requirements review as the quality gate — our requirements package must pass the requirements review to be accepted for use in the production of a design.